Modeling Web-Based Dialog Flows for Automatic Dialog Control

Matthias Book, Volker Gruhn
Chair of Applied Telematics / e-Business, Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany; Phone: +49-341-97-32330, Fax: +49-341-97-32339
{book, gruhn}@ebus.informatik.uni-leipzig.de

Abstract

In web-based applications, the dialog control logic is often hidden in or entwined with the presentation and/or application logic, even if the latter tiers are well-separated. This makes it difficult to control complex dialog structures like nested dialogs, and to reconcile the device-independent business logic with the device-specific interaction patterns required by different clients' I/O capabilities. To avoid continuous re-implementation of the dialog control logic, we present a Dialog Control Framework that is separate from the presentation and business tiers, and manages arbitrarily nested dialog flows on different presentation channels. The framework relies on dialog specifications developed using the Dialog Flow Notation, which are translated into an object-oriented dialog flow model for efficient run-time lookups. This way, the framework automates the dialog control aspect of web-based application development and leaves only the tasks of implementing the business logic, designing the hypertext pages, and specifying the dialog flow to the developer.

1. Introduction

Over the past years, business processes have become increasingly distributed in character and recently even begun to exhibit mobile aspects, especially in application areas such as field sales forces, logistics infrastructure, etc. Today, users demand flexible access to services — ideally, any application should be available on any device, anywhere, anytime [19]. Since economic considerations forbid implementing applications individually for every kind of device, and mobile devices typically have strict energy, memory, input and output limitations [12], the users' demands can virtually only be fulfilled by pursuing a thin-client approach [18]. Web-based applications seem to be ideal implementations of this concept, since the complete business logic resides on a central server, while the user interface (UI) consists entirely of web pages or similar renderings on client devices such as desktop PCs, PDAs, mobile phones etc. [8]

However, the I/O capabilities of these devices range widely, and characteristics like screen size do not only impact the page layout, but also affect how users work with an application [4]: A dialog that may be completed in a single step on a desktop browser may have to be broken up into multiple interaction steps on a mobile device whose small screen cannot accommodate large forms. However, the server-side business logic should remain independent of such client-side specifics. This obviously calls for separating the device-specific presentation from the device-independent business logic — however, as we will show in section 2, that is not as trivial as it sounds since the dialog control logic tends to get mixed up with either one of those tiers.

Another challenge when developing and working with web-based instead of window-based UIs is induced by the page-based interface paradigm [21]: In window-based applications, any window can spawn "child windows", and the completion of a dialog in a child window returns the user to the dialog in the parent window. Users can rely on this predictable behavior that reinforces their conceptual model and thus increases applications' usability [15]. In web-based applications, however, the page-based presentation and the stateless request-response communication protocol complicate the control of the dialog structure: Since the business logic cannot push data to the client, it can only react passively to user actions (e.g. clicking on a link) instead of actively initiating dialog steps (e.g. opening a new window). Also, the Hypertext Transfer Protocol (HTTP) only transports data, but does not maintain any state information. Consequently, the application itself has to manage the dialog state for each user session. While simple linear and branched dialog structures can be implemented with basic session state management techniques, arbitrarily nested dialogs require more complex dialog control logic to keep track of users' dialog state on the server. To avoid this effort, most of today's web-based applications do not offer nested dialogs. Instead, users have to navigate between parent and

child dialogs manually (if such a hierarchical relationship is distinguishable at all). However, this does not conform to the conceptual model of nested dialogs that users have long established through window-based applications — a violation of the ISO dialog principles of controllability and conformity with user expectations [1] that imposes a high cognitive and memory load on the user and thus reduces web-based applications' usability.

Since these issues of device-dependent interaction patterns and nestable dialogs are independent of specific applications, they should be addressed by generic solutions. We therefore present the architecture of a Dialog Control Framework capable of managing complex, nested dialog flows on different devices (section 2). We also introduce a graphical Dialog Flow Notation (section 3) for the specification of such flows, and finally show how they can be modeled in a way that allows the framework to interpret them efficiently (section 4). We hypothesize that the use of these tools will increase the efficiency of software development processes for web-based applications, as well as their usability.

2. Dialog Control Framework

Web-based applications are usually designed according to the Model-View-Controller (MVC) paradigm [13], which suggests the separation of user interface, business logic and control logic. While user interface and business logic can be distinguished quite naturally ("what the user sees" vs. "what the system does"), the distinction between business logic and dialog control logic is much more subtle ("what the system does" vs. "what it should do next, based on the user's input"). Therefore, it is easy to mix up the implementation of application and dialog control logic, even if both are separated well from the presentation logic.

For example, in the Apache Jakarta Struts framework [2], the dialog flow is controlled by so-called actions. They implement the application logic and also decide where to forward a request, while the controller just executes that forward command. As indicated by the shading in Fig. 1, the dialog control logic is distributed over all actions in this approach, i.e. the dialog flow is not specified outside the business logic, but actually implemented in the Java code of the actions. This allows the actions to make only relatively isolated dialog flow decisions, and hampers the implementation of more complex dialog structures with constructs like nested dialog modules. To raise the actions' awareness of the "big picture" and enable them to control more complex constructs, still more control logic would have to be implemented in them, exacerbating the problem. Also, the hard-coded decentralized implementation of the dialog control logic is relatively inflexible, almost unsuitable for reuse and hard to maintain. Finally, achieving device independence would require additional effort and possibly redundant work: Since the dialog flow depends on the presentation channel, while the business logic should not, their close coupling prevents the reuse of actions on multiple presentation channels. Instead, each channel would require its own set of actions to implement the individual dialog flow for the respective devices.

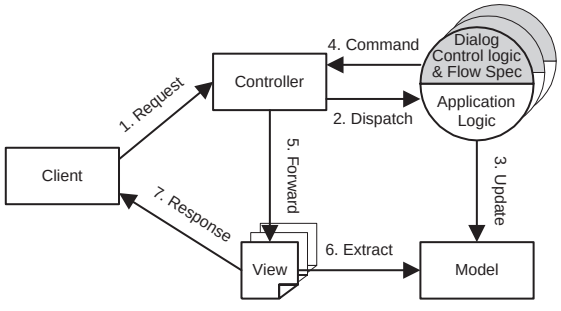

Figure 1. Coarse architecture of the Struts framework

In contrast, the Dialog Control Framework (DCF) presented in this paper features a very strict implementation of the MVC pattern, completely separating not only the business logic and UI, but also the dialog flow specification and dialog control logic: The controller decides where to forward requests by using a central dialog flow model to look up the receivers of events generated by masks and actions.

As the coarse architecture (Fig. 2) shows, the actions are relatively lightweight here since they contain only calls to the business logic, while all dialog control logic has been moved to the dialog controller. This controller does not receive requests from the clients directly anymore. Instead, on each presentation channel, it receives events that have been extracted from the requests by channel servlets. The dialog controller looks up the receivers of these events in the dialog flow model — a collection of objects representing dialog elements that hold references to each other to mirror the dialog flow (section 4). This dialog flow model is built upon initialization of the framework by parsing documents containing the dialog flow specification in an XML-based format (the shaded parts of the diagram emphasize that the dialog control logic and the flow specification are decoupled from the business logic and from each other in this approach). Depending on the receiver that the controller retrieved from the model for an event, it may call an action, forward the request to a mask, or nest or terminate dialog compounds (a construct introduced below). The latter operations are performed on compound stacks that store the nested compounds constituting the state of the dialog system for each user.

This centralized approach to dialog control has three ad-

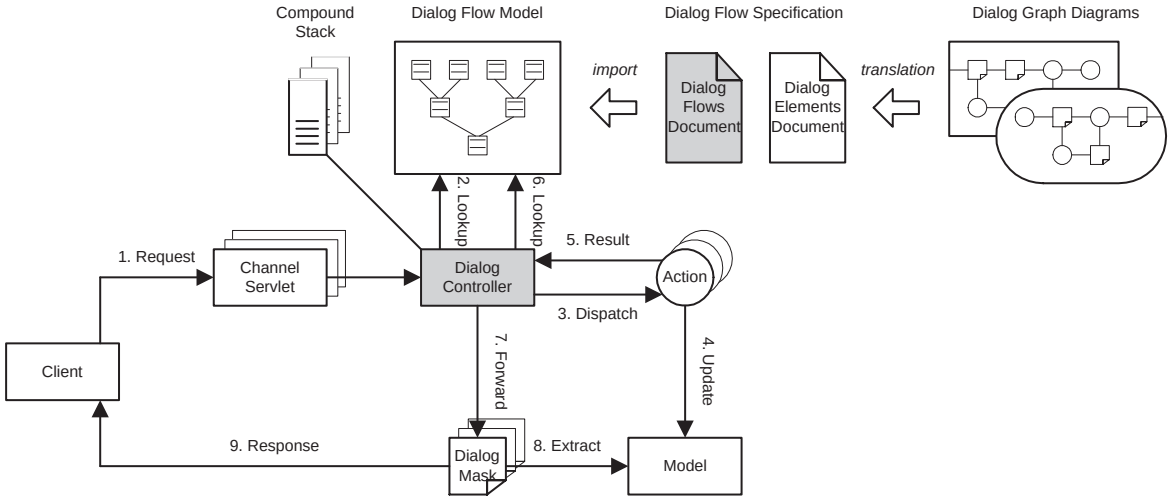

Figure 2. Coarse architecture of the DCF

vantages over the previously discussed decentralized architecture: Firstly, the strict separation between business logic implementation, UI design, dialog flow specification and dialog control logic enables a high degree of flexibility, reusability and maintainability for the components of all four tiers. Secondly, due to this clean separation, device-independent applications can be built with minimal redundancy: Only the dialog masks and the dialog flow specifications need to be specified for the different presentation channels, while the business logic is implemented device-independently only once and the dialog control logic is provided by the framework. Finally, since the central dialog control logic is aware of the whole dialog flow specified for each channel (it knows the "big picture"), it can manage complex dialog constructs.

The framework was implemented using Java 2 Enterprise Edition. To build an application with it, developers only need to provide simple Java classes implementing the actions, JavaServer Pages implementing the dialog masks, and documents containing the dialog flow specification. Written in the XML-based Dialog Flow Specification Language (DFSL), these documents are machine-readable representations of dialog graph diagrams drawn in the Dialog Flow Notation presented next. Since these deliverables are completely application-specific, the framework is suitable for black box reuse.

3. Dialog Flow Notation

The Dialog Flow Notation (DFN) represents the dialog flow within an application as a directed graph of states connected by transitions. To define the concept of a "dialog flow" and develop the notation elements, we first examine the client-server communication taking place in each request-response cycle of a hypertext-based application:

As Fig. 3 shows, a page *A'* displayed on the client is rendered from source code (e.g. HTML) that was first generated by an entity *A* (e.g. a JavaServer Page) on the server and then transmitted to the client. When the user follows a link or submits a form on this page, the resulting data *a* is transmitted to the server. The application logic may now process the data in a number of steps (here: *1* and *2*), which each generate data (*b* and *c*) that is processed in the next step. Finally, the source code for the following page is generated (*B*), transmitted to the client and rendered there (*B'*). Alternatively, user-submitted data (such as *d*) may not require any application logic processing, but directly lead to the generation and rendering of a new page (*C* and *C'*). We call the server activity happening between the submission of a request and the receipt of a response by the client a **dialog step**, and refer to all possible sequences of dialog steps as an application's **dialog flow**.

Looking at the communication model in Fig. 3, we realize that the client-server communication and thus the dis-

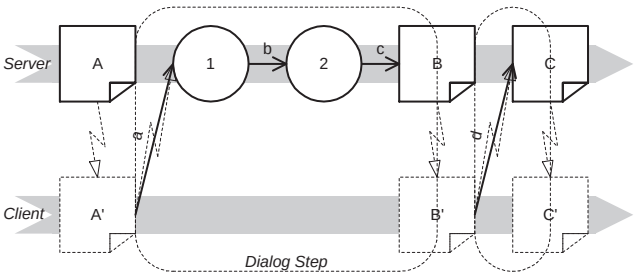

Figure 3. Request-response cycles in HTTP

tinction between generating (*A*) and rendering pages (*A'*) is irrelevant for the purpose of modeling dialog flows: When specifying how the user interacts with the application logic via the UI pages, the dialog flow designer does not need to know about technical details such as pages' source code being generated on the server and transmitted to the client prior to rendering. The DFN therefore only specifies the order of the UI pages and processing steps, and the data exchanged between them. It models the dialog flow as a transition network, i.e. a directed graph of states connected by transitions called a **dialog graph**. The notation refers to the transitions as **events** and to the states as **dialog elements**, discerning atomic and compound elements.

3.1. Basic dialog elements and events

Hypertext pages (symbolized by dog-eared sheets and referred to by the more generic term **masks** here) and application logic operations (symbolized by circles and called **actions** from now on) constitute the **atomic dialog elements**. As illustrated in the communication model in Fig. 3, some dialog sequences may contain multiple consecutive masks or actions, so dialog graphs do not need to be bipartite. Every dialog element can generate and receive multiple events, enabling the developer to specify much more complex dialog graphs than the linear succession of elements shown above. Which element will receive an event depends both on the event and the generating element (e.g., an event *e* may be received by action *3* if it was generated by mask *D*, but be received by action *4* if generated by mask *E*). Events can carry parameters, i.e. application-specific information such as form input, and thus facilitate communication between dialog elements.

Theoretically, the complete dialog flow of an application could be described using only atomic elements. However, the resulting specification would be too complicated to understand, and the "flat" structure does not support reuse of often-needed dialog graphs. The DFN therefore provides **compound dialog elements** (compounds) which encapsulate dialog graphs and allow the nesting of dialog structures: A compound's interior dialog graph can contain subcompounds, and the compound itself can be embedded in the exterior dialog graphs of super-compounds. We discern two types of compound dialog elements: **Dialog modules** (symbolized by boxes with rounded corners) contain an interior dialog graph with one entry point and one or more exit points, while **dialog containers** (symbolized by boxes with right-angled corners) contain an interior dialog graph with one entry point, but no exit points. At the top of the nesting hierarchy is an **application container** (symbolized by a double-line box) that is entered when the client sends the initial request to the server running the application, and can only be exited by leaving the site.

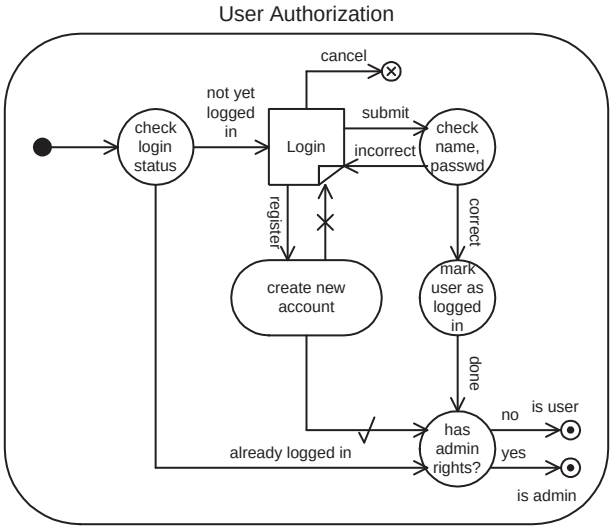

Figure 4. *User Authorization* dialog module

We will introduce the features of dialog modules using the *User Authorization* module in Fig. 4 as an example. This module checks if the user is already logged in and shows a *Login* mask to prompt for his user name and password, if necessary. If the user's credentials are correct, the module marks him as logged in, checks his access rights and terminates, notifying the super-compound of the user's status. If the user does not yet have an account, he can register using the embedded *create new account* sub-module. Note that by splitting the application logic into fine-grained operations instead of implementing them all together, the module can react flexibly to different situations, like bypassing the credential check when the user is already logged in.

When a compound receives an event from the exterior dialog graph that it is embedded in, traversal of its interior dialog graph starts with the **initial event**. When the interior dialog graph terminates, it generates a terminal event that is propagated to the super-compound and continues the traversal of the exterior dialog graph. Depending on the semantics of the termination, developers can choose between three kinds of terminal events (Fig. 5).

Regular terminal events are intended to communicate application-specific information to the terminating module's exterior dialog graph, such as the result of an operation or decision (for example, the *User Authorization* module generates an *is user* or *is admin* terminal event, depending on the user's rights). Often, however, modules do not need to notify their calling super-compound about some application-specific state, but should simply indicate if they completed their task successfully or not. The DFN provides the ***done*** and ***cancelled*** **terminal events** to model these situations (for example, the *create new account* module may terminate with a *done* or *cancelled* event, depending on the

Event type	Interior dialog graph symbol	Exterior dialog graph symbol
Initial event		*n/a*
Regular terminal event		
Done terminal event		
Cancelled terminal event		
Abort event		*n/a*

Figure 5. Event types and notation symbols

success of the registration process). In contrast to regular terminal events, *done* and *cancelled* events are unnamed and cannot carry parameters. Their universal, application-independent semantics enable the dialog control logic to handle them automatically in situations where a receiver could not be specified at design-time, but needs to be identified at run-time. This is often the case when using the compound or common events described next.

3.2. Advanced dialog constructs

Complex dialog structures will usually contain a certain amount of redundancy, since some dialog elements may be linked from many other elements in the application. If we had to specify all the respective events explicitly, our dialog graph diagrams would soon become cluttered with redundant information. To counter the combinatory explosion of transitions that often plagues state machines, Harel's Statecharts [10] provide the construct of a transition leading from a contour to a state. The DFN uses a similar construct, albeit adapted for dialog flow specification: A so-called **compound event**, symbolized in dialog graph diagrams by an arrow leading from the compound's contour to a certain element, indicates that this event may be generated by every element in the compound.

As an example, consider the dialog graph of a simple online shop in Fig. 6. The shop's homepage, list of items in each category, detailed description of each item, shopping cart and checkout process shall be linked from every mask in the system. If all events connecting these elements had been specified explicitly, a tangled event web would have been the result. Using compound events, however, we can express the same relationships in a much clearer diagram.

The scope of compound events only encompasses the compound that they are specified in, but not its super- or sub-compounds. For example, while the *show item* event leads to the *Item Details* mask from any mask in the *Shop* container, such a connection does not exist for any masks inside the *Checkout* sub-module. In some situations, however, it may actually be desirable that certain events can be handled even if their receiver is not specified in the compound that they are generated in — for example, the *create new account* module may be reachable from anywhere within a web-based application, not just from the *Login* mask. To model these relationships, the DFN provides the **common event**. Similar to the compound event, it is symbolized by an arrow leading away from the compound's contour, but outward to another compound element (and only to a compound — it may not lead to an atomic element or into a dialog graph). This so-called **common compound** is then nested into the user's dialog sequence wherever he generates the respective common event, independently of his position in the dialog flow.

As an example, consider the *Portal* application container in Fig. 7 (the application container, symbolized by a double-line box, is the root of the compounds' nesting hierarchy, where every user's dialog sequence starts when he enters the application). The parts of this portal system are modeled as common compounds so they can be reached from anywhere within the application.

Compound and common events provide a mechanism for entering compounds from different places in a dialog flow without having to specify the respective events explicitly. For usability reasons, we would want to return the user to the mask from which he had entered the compound when

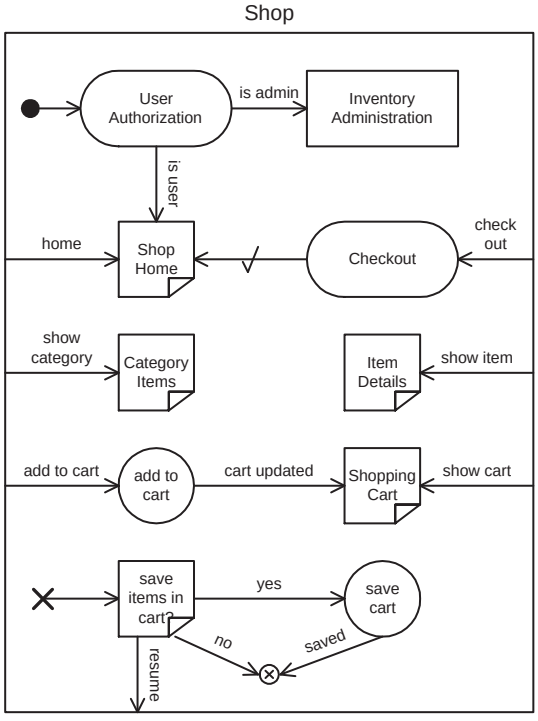

Figure 6. Dialog graph of *Shop* container

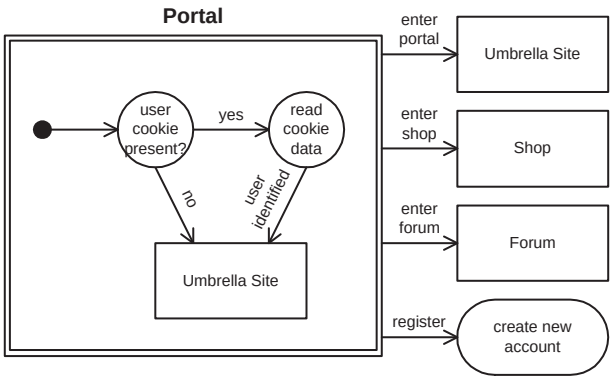

Figure 7. Dialog graph of *Portal* application container

it terminates (in the same way that window-based applications return the focus to the parent window after the user closed a child window). However, since we do not know at specification time where to return the user, we cannot specify the receiver of the terminal event. The DCF introduced in section 2 solves this apparent dilemma by using the *done* and *cancelled* events' application-independent semantics described above: If the framework intercepts a *done* or *cancelled* event without a specified receiver, the so-called **return mechanism** automatically leads the event to the dialog mask from which the terminated module was activated, creating the familiar "nesting" effect for the user.

This mechanism works for modules, which by definition terminate eventually. However, common containers pose more of a challenge: Since they do not terminate by themselves, and nesting them deeper and deeper into each other as the user navigates between them would gradually lock up memory, the only option is to abort a common container before another one can be activated at the same nesting level. For example, if the user is currently in the *Shop* container and generates an *enter portal* event, traversal of the *Shop* container's interior dialog graph (and of all compounds nested into it at the time) has to be aborted before the *Umbrella Site* container's initial event can be handled.

In order to abort a compound in a controlled way, a special **abort dialog graph** can be specified for it, which might ask the user if he really wants to abort (also giving him a chance to resume the original dialog graph where he left off), or if he wants to save any unsaved data before aborting. Traversal of the abort dialog graph, which may not contain any sub-compounds and must not be connected to the compound's regular dialog graph, starts at the **abort event** (see symbol in Fig. 5). For example, in the *Shop* container's abort dialog graph (at the bottom of Fig. 6), the system prompts the user if he wants to save the items in his cart before leaving the shop, or if he wants to resume shopping. In case the user decides not to switch containers, he can generate a **resume event** (symbolized in dialog graph diagrams by an arrow leading towards the compound's contour), which invokes the framework's **resume mechanism**. Using an algorithm similar to the return mechanism, it leads the user back to the dialog mask in the regular dialog graph that was displayed before the abort sequence started.

3.3. Presentation channels

To specify different dialog flows for different devices, the flows for each channel are specified in separate compounds with **channel labels** added after the compounds' names. For example, Fig. 8 specifies the dialog flows for a *Checkout* module on the HTML and WML channel. Note that while the channels employ different dialog masks according to the clients' input/output capabilities, they use the same actions for processing the users' input. This enables developers to implement the device-independent application logic only once and then reuse it on multiple channels. In addition to specifying presentation channel-specific dialog graphs for different devices, it is also possible to specify generic, presentation channel-independent aspects of the dialog flow only once and reuse them on multiple channels in order to reduce redundant specifications.

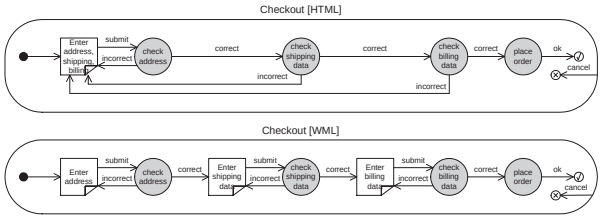

Figure 8. *Checkout* module on HTML and WML presentation channel

4. Dialog Flow Model

For a smooth transition from specification to implementation, the graphical dialog flow specifications are translated into machine-readable documents written in the XML-based DFSL (DFSL). As described in section 2, the dialog controller must constantly look up the receivers of certain events generated by certain elements at run-time of the application. Since it would be very time and memory consuming to read the DFSL documents in each dialog step, parse the required parts and instantiate the respective dialog elements, the documents are transformed into an object representation upon initialization of the framework. The framework then works with this object representation — the **dia-**

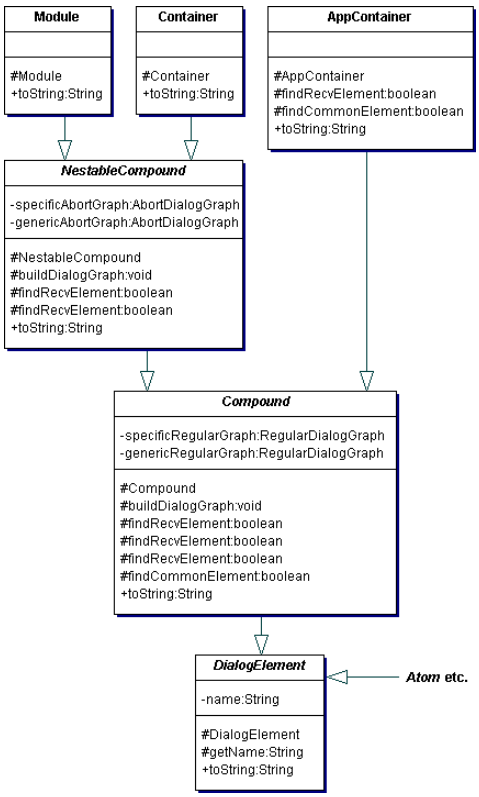

Figure 9. Dialog elements class diagram

log flow model — directly without costly and cumbersome conversions.

4.1. Dialog elements

The dialog flow model is a set of objects (representing the application's dialog elements) that maintain a network of references to each other in order to specify which elements generate and receive which events. In an excerpt from the framework's class diagramm, Fig. 9 shows the inheritance structure of compound dialog elements. In analogy to the DFN, a `DialogElement` can either be a `Compound` or an `Atom`. We discern three types of compounds: The `AppContainer`, which cannot be nested into any other compound, and the `NestableCompound` types of `Module`s and `Container`s.

We do not discuss atomic dialog elements further here since their representation in the dialog flow model is comparatively trivial. The `Compound` class, however, is quite a bit more complex than the other classes in the model. It contains a compound's regular dialog graph (traversed when the compound is not aborting) in the form of two `RegularDialogGraph`s: The `specificRegularGraph` contains the presentation channel-specific parts of the dialog graph, while the `genericRegularGraph` contains the device-independent parts of the graph. In addition, `NestableCompound`s contain a `specificAbortGraph` and `genericAbortGraph` which together contain the compound's complete abort dialog graph. Read access to this model is provided by the `findRecvElement` and `findCommonElement` methods that return the receiving elements for the given events.

4.2. Dialog graphs

`RegularDialogGraph`s and `AbortDialogGraph`s are sub-classes of the abstract `DialogGraph` class, as shown in the class diagram excerpt in Fig. 10. They contain the actual dialog graphs, modeled by event tables which hold references to the dialog elements that receive the events that may be generated in a compound: The `maskEventTables` and `actionEventTables` contain entries for each mask and action in a `DialogGraph`. Each entry is an `EventTable`, which in turn contains entries for each event generated by that element. Each of those entries, finally, contains a reference to the receiving `DialogElement`. Fig. 11 visualizes the structure of these nested tables, with key-value-relationships in hash table entries being visualized by two columns in a box.

Since regular dialog graphs may not only con-

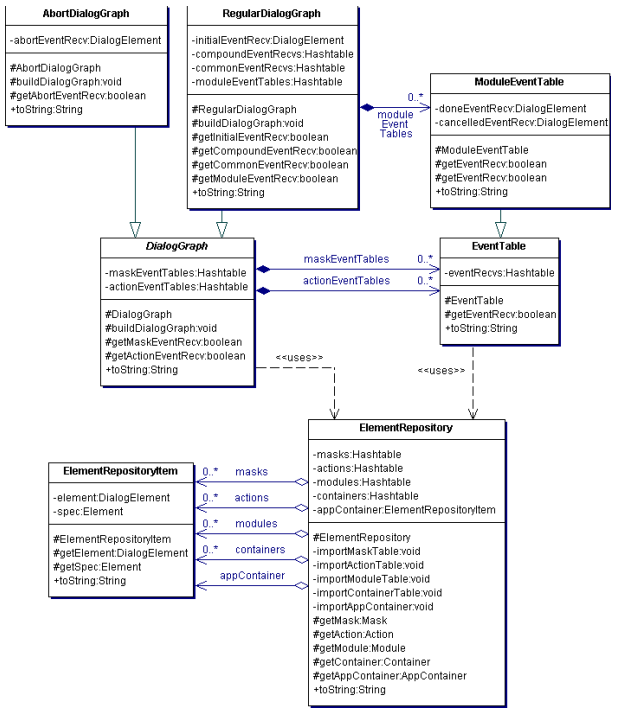

Figure 10. Dialog graphs class diagram

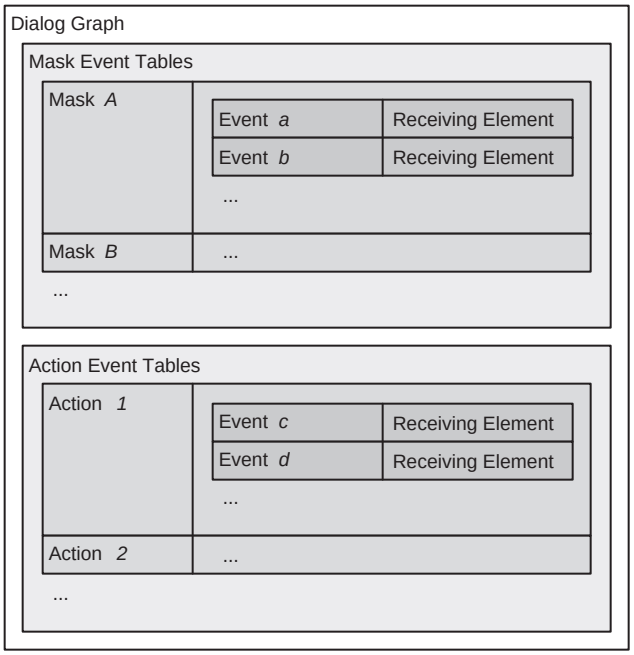

Figure 11. Event table structure for dialog graphs

tain masks and actions, but also modules, the `RegularDialogGraph` sub-class adds the `moduleEventTables` attribute which contains the tables for events generated by modules. As terminating modules may generate *done* and *cancelled* events in addition to regular events, the `ModuleEventTable` class extends the `EventTable` class to add the `doneEventRecv` and `cancelledEventRecv` attributes, which store the receivers of the respective events. Besides modules, regular dialog graphs also contain an initial event, whose receiving dialog element is stored directly in the `initialEventRecv` attribute. They may also contain compound and common events, whose receivers are stored in the `compoundEventRecvs` and `commonEventRecvs` hash tables. The complete event table structure for `RegularDialogGraph`s is visualized in Fig. 12.

Nestable compounds, finally, may not only contain a regular dialog graph, but also an abort dialog graph which is traversed upon abortion of the compound. Since this dialog graph may not contain any modules, compound or common events, the `AbortDialogGraph` sub-class modeling it only adds the `abortEventRecv` attribute which contains a reference to the dialog element receiving the abort event. The mask and action event tables for the abort dialog graph are inherited from the `DialogGraph` super-class, as visualized in Fig. 12. Read

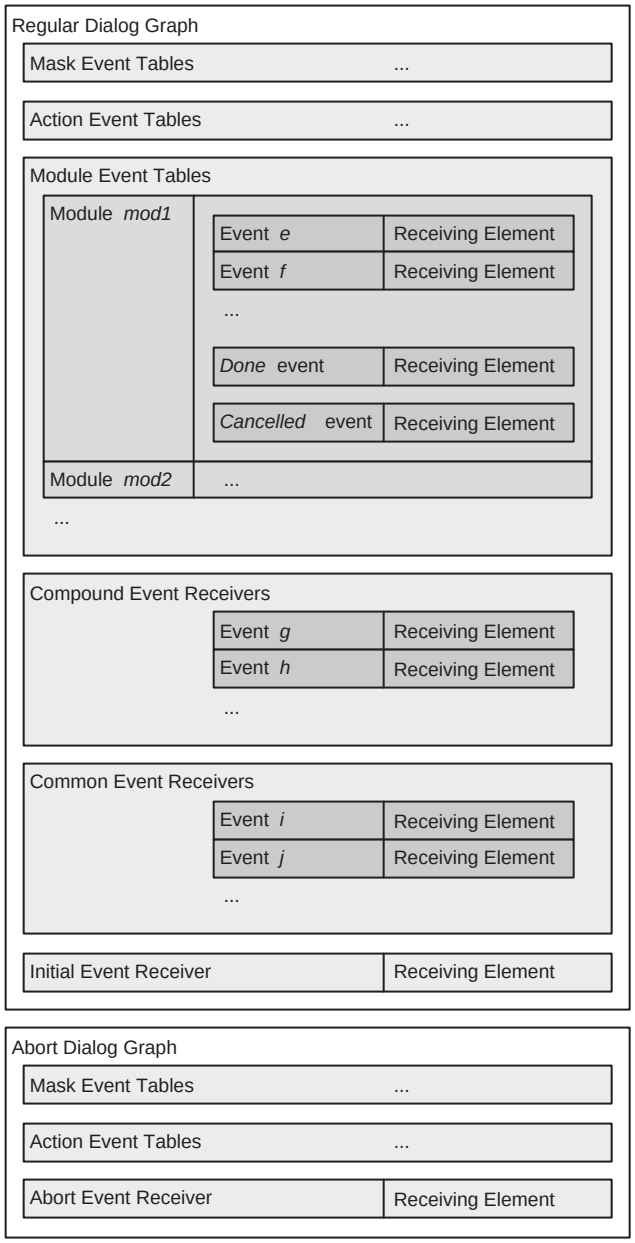

Figure 12. Event table structure for regular and abort dialog graphs

access to the event tables is provided by the classes' various `get...EventRecv` methods which are called by the compounds' `findRecvElement` methods.

When a user is working with the application, the `Module`, `Container` and `AppContainer` objects that he encounters are stored on his compound stack (Fig. 2) in order to keep track of the nested dialog structure and allow direct access to the dialog graph of the currently traversed compound.

5. Related Work

Most tools offering dialog control implementation support for web-based applications follow the MVC design pattern to facilitate easier dialog control. The Apache Jakarta Project's Struts framework [2] is the most popular solution today, however, it forces developers to combine business logic and dialog control logic in the actions (as discussed in section 2), which renders the dialog control implementation cumbersome and inflexible. While the concept of an application-independent "screen flow manager" that determines the next view is suggested in the Java BluePrints [17], no framework seems to exist yet that employs this pattern to implement complex dialog constructs such as the arbitrarily nestable, abortable and resumable dialog modules and device-specific dialog flows offered by the DCF.

The challenges posed by different devices' interaction patterns are addressed in the Sisl (Several Interfaces, Single Logic) approach [3]. Its "service monitor" can process unordered or incomplete input from a wide range of client devices. However, since it uses acyclic graphs to model dialogs, it is more suitable for simple linear and branched dialog structures than for highly interactive applications with nested and cyclic dialogs. Similarly, the World Wide Web Consortium's XForms initiative [20] is also concerned with the device-independent specification of forms. However, it is mostly concerned with the specification of widgets (such as input fields) on web pages and does not allow the specification of nested dialog sequences.

For the specification of user interfaces, a number of notations have been proposed over time. However, approaches that were explicitly developed for web-based systems mostly focus on data-intensive information systems, but not interaction-intensive applications [6]: For example, the RMM development process [11] allows the definition of navigable relationships between data entities, and the OOHDM [16] process provides classes like node, link and index to represent different forms of navigation; however, the resulting structures remain "flat" and cannot be nested arbitrarily. The same is true for the HDM-lite notation used by the Autoweb tool [7], which supports the automatic generation of database schemas and application pages from a conceptual model. Finally, while the language WebML [5] is capable of modeling simple dynamic features of a data-intensive web application by providing operation units for creating, deleting and modifying entities, it does not support more complex structures such as modular, nestable dialog sequences.

While the concept of modeling dialog systems as state-based systems is not new [9] and generic notations for this already exist (e.g. Statecharts [10]), we chose not to use any generic notation because expressing the particularities of web-based dialog flows (e.g. different types of dialog elements, compounds and events) in those would be cumbersome in practice. Also, we wanted to provide the DFN with constructive instead of mere descriptive power, enabling developers to use complex constructs like the abort/resume mechanism intuitively, without having to spell out their details in a generic notation. In consequence, the notation contains a number of elements that may seem like "syntactic sugar" at first glance, but should actually strengthen the notation and framework's applicability to common challenges that web engineers face in real-world projects.

6. Conclusions

By comparing the characteristics of window- and web-based user interfaces, we found that the latter's usability suffers from a lack of support for nested dialogs and the difficulty of reconciling the device-independent business logic with the device-dependent interaction patterns required by different clients' I/O capabilities. In order to avoid re-implementing the complex dialog control logic in every application, we presented a Dialog Control Framework (section 2) for automatic dialog control in web-based applications. To control users' interaction with an application, the framework requires a specification of the application's dialog flow. This specification is first developed in the graphical Dialog Flow Notation (section 3) and then translated into the XML-based Dialog Flow Specification Language. After parsing these specifications, the framework creates an object-oriented dialog flow model (section 4) that represents the dialog structure as a graph, enabling efficient access to all dialog elements.

From a practical perspective, a weak point of the notation may currently be the fine granularity of actions that is required to reuse them flexibly in dialog graphs on different presentation channels (this especially concerns actions responsible for processing user input submitted through forms): The finer the actions are grained, the easier it is to adapt to different interaction patterns — however, very fine granularity also results in quite high specification, implementation and performance overhead. When specifying a dialog flow, the developer therefore needs to find a balance between the desired flexibility and the required granularity. Research on solutions to this dilemma is in progress

— a possible approach seems to be abstracting from concrete masks and actions, and just letting the developer specify which data the user shall be prompted for. The framework would then have to generate a suitable dialog flow for obtaining and checking the user input automatically based on the specifications.

While the implementation of the framework and the structure of the object-oriented dialog flow model already define operational semantics for the notation, a formal definition of the DFN's semantics must obviously be established as a sound basis for future work. This can be achieved by showing that all DFN constructs can also be expressed by means of a more generic formalism such as Statecharts or Petri nets (even if that formal representation would not be suitable for practical use). We are currently working on the definition of such a formal basis that will enable us to reason about the specifications produced with the DFN. Also, to enable an automatic transition from the graphical to the machine-readable dialog flow specification, we are currently defining a dialog flow metamodel based on the Object Modeling Groups Meta-Object Facility [14], and developing a plug-in for the open source Eclipse IDE to model dialog flows graphically in the DFN and create DFSL documents out of them automatically by applying XSLT style sheets to their XMI representation.

More empiric evidence is still needed to see how the Dialog Control Framework and Dialog Flow Notation can be integrated into the software development process for web-based applications. To validate the suitability of both tools, a small-scale demo application employing all dialog control features was already developed at the Chair of Applied Telematics' Mobile Technology Lab. We are currently striving to gain more experience from larger projects, which should yield insights into the applicability of the notation and framework to certain application domains and client devices, and enable us to evaluate the increase in development efficiency and application usability gained by the use of both tools.

7. Acknowledgments

The Chair of Applied Telematics/e-Business at the University of Leipzig is endowed by Deutsche Telekom AG.

References

[1] Ergonomic requirements for office work with visual display terminals (VDTs) — Part 10: Dialogue principles. Technical Report ISO 9241-10, International Organization for Standardization, 1996.

[2] Apache Jakarta Project. Struts. http://jakarta.apache.org/struts/.

[3] T. Ball, C. Colby, and P. Danielsen. Sisl: Several interfaces, single logic. *International Journal of Speech Technology*, 3(2):91–106, 2000.

[4] M. Butler, F. Giannetti, R. Gimson, and T. Wiley. Device independence and the web. *IEEE Computing*, 6(5):81–86, Sep–Oct 2002.

[5] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): A modeling language for designing web sites. *Computer Networks*, 33:137–157, Jun 1995.

[6] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, Sep 1999.

[7] P. Fraternali and P. Paolini. Model-driven development of web applications: The Autoweb system. *ACM Transactions on Information Systems*, 28(4):323–382, Oct 2000.

[8] M. Gaedke, M. Beigl, H. Gellersen, and C. Segor. Web content delivery to heterogeneous mobile platforms. *Advances in Database Technologies, Lecture Notes in Computer Science*, 1552, 1998.

[9] M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, Jul 1986.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] T. Isakowitz, E. Stohr, and P. Balasubramanian. RMM: a methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, Aug 1995.

[12] J. Jing, A. Helal, and A. Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(6):117–157, Jun 1999.

[13] G. Krasner. A cookbook for using the model-view-controller user interface paradigm in Smalltalk. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[14] Object Management Group. Meta-object facility (MOF), v1.4. http://www.omg.org/technology/documents/formal/mof.htm, Apr 2002.

[15] J. Rice, A. Farquhar, P. Piernot, and T. Gruber. Using the web instead of a window system. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '96)*, 1996.

[16] D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Communications of the ACM*, 38(8):45–46, Aug 1995.

[17] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, 2nd edition, 2002.

[18] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7):77–98, Jul 1992.

[19] M. Weiser. Computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7), Jul 1993.

[20] World Wide Web Consortium. XForms 1.0, W3C recommendation. http://www.w3.org/TR/2003/REC-xforms-20031014/, Oct 2003.

[21] W. Zhao, D. Kearney, and G. Gioiosa. Architectures for web based applications. In *4th Australasian Workshop on Software and Systems Architectures (AWSA 2002)*, 2002.